Exercice 1 (parcours de tableaux) :

1. Écrire une fonction `somme : int array -> int` prenant en argument un tableau d'entiers et renvoyant la somme des éléments de ce tableau.

Exemple

```
# somme [|1;2;3;4;5;6|] ;;
- : int = 21
```

2. Écrire une fonction `moyenne : int array -> float` prenant en argument un tableau d'entiers et renvoyant la moyenne des éléments de ce tableau.

Exemple

```
# moyenne [|1;2;3;4;5;6|] ;;
- : float = 3.5
```

3. Écrire une fonction `min_tab : 'a array -> 'a` prenant en argument un tableau et renvoyant la plus petite valeur qu'il contient. On renverra une erreur si le tableau est vide.

Exemple

```
# min_tab [|1;2;3;4;0;5;1;6|] ;;
- : int = 0
```

4. Écrire une fonction `max_tab : 'a array -> 'a` prenant en argument un tableau et renvoyant la plus grande valeur qu'il contient. On renverra une erreur si le tableau est vide.

Exemple

```
# max_tab [|1;2;3;4;0;5;1;6|] ;;
- : int = 6
```

5. Écrire une fonction `appartient : 'a -> 'a array -> bool` prenant en arguments un élément `x` et un tableau `t`, et renvoyant `true` si `x` est présent dans `t` (et `false` sinon).

Exemple

```
# appartient 0 [|1;2;3;4;0;5;1;6|] ;;
- : bool = true
# appartient 9 [|1;2;3;4;0;5;1;6|] ;;
- : bool = false
```

6. Écrire une fonction `dernier_indice : 'a -> 'a array -> int option` prenant en arguments un élément `x` et un tableau `t`, et renvoyant `Some i` où `i` est l'indice de la dernière occurrence de `x` dans `t`. On reverra `None` si `x` n'est pas présent dans `t`.

Exemple

```
# dernier_indice 1 [|2;3;1;4;0;5;1;6|] ;;
- : int option = Some 6
# dernier_indice 9 [|2;3;1;4;0;5;1;6|] ;;
- : int option = None
```

7. Écrire une fonction `premier_indice : 'a -> 'a array -> int option` prenant en arguments un élément `x` et un tableau `t`, et renvoyant `Some i` où `i` est l'indice de la première occurrence de `x` dans `t`. On reverra `None` si `x` n'est pas présent dans `t`.

Exemple

```
# premier_indice 1 [|2;3;1;4;0;5;1;6|] ;;
- : int option = Some 2
# premier_indice 9 [|2;3;1;4;0;5;1;6|] ;;
- : int option = None
```

8. Écrire une fonction `forall : ('a -> bool) -> 'a array -> bool` prenant en arguments une fonction f de type `'a -> bool` et un tableau t, et renvoyant `true` si $\forall x \in t, f(x)$ est vrai (et `false` sinon).

Exemple
```
# forall (fun x -> x mod 2 = 0) [|2;3;1;4;0;5;1;6|] ;;
- : bool = false
# forall (fun x -> x < 9) [|2;3;1;4;0;5;1;6|] ;;
- : bool = true
```

9. Écrire une fonction `exists : ('a -> bool) -> 'a array -> bool` prenant en arguments une fonction f de type `'a -> bool` et un tableau t, et renvoyant `true` si $\exists x \in t$ tel que $f(x)$ est vrai (et `false` sinon).

Exemple
```
# exists (fun x -> x mod 2 = 0) [|2;3;1;4;0;5;1;6|] ;;
- : bool = true
# exists (fun x -> x > 9) [|2;3;1;4;0;5;1;6|] ;;
- : bool = false
```

Exercice 2 (exceptions) :
Si ce n'est pas déjà le cas, modifier vos fonctions `appartient`, `premier_indice`, `for_all` et `exists` pour qu'elles s'arrêtent dès que poossible. On pourra déclarer et utiliser les exceptions suivantes :

```
exception Break
exception Trouve of int
```

Exercice 3 (création de tableaux) :

1. Écrire une fonction `carres : int -> int array` prenant en argument un entier n et renvoyant un tableau `t` de taille n tel que $\forall i$, `t.(i)` contienne la valeur i^2.

Exemple
```
# carres 6 ;;
- : int array = [|0; 1; 4; 9; 16; 25|]
```

2. Écrire une fonction `copie_miroir : 'a array -> 'a array` prenant en argument un tableau `t` et renvoyant un nouveau tableau contenant les mêmes éléments que `t` mais dans l'ordre inverse.

Exemple
```
# copie_miroir [|"a";"b";"c";"d"|] ;;
- : string array = [|"d"; "c"; "b"; "a"|]
```

3. Écrire une fonction `matrice : int -> int array array` prenant en argument un entier n et renvoyant une matrice M de taille $n \times n$ tel que $\forall (i,j),\ M_{i,j} = n \times i + j$.

Exemple
```
# matrice 6 ;;
- : int array array =
[|[|0; 1; 2; 3; 4; 5|];
  [|6; 7; 8; 9; 10; 11|];
  [|12; 13; 14; 15; 16; 17|];
  [|18; 19; 20; 21; 22; 23|];
  [|24; 25; 26; 27; 28; 29|];
  [|30; 31; 32; 33; 34; 35|]|]
```

Exercice 4 (tri à bulles) :
Le **tri à bulles** est un algorithme qui permet de trier un tableau d'éléments dans l'ordre croissant. Il repose sur le principe suivant :

(a) on parcourt le tableau de gauche à droite, en comparant les éléments consécutifs : s'ils ne sont pas dans le bon ordre, on les échange.

(b) à la fin du parcours :

- si on a effectué au moins un échange durant ce parcours, on recommence l'étape (a) ;
- sinon, le tableau est trié.

1. Écrire une fonction `swap : 'a array -> int -> int -> unit` telle que `swap t i j` échange les valeurs des cases `t.(i)` et `t.(j)`.
2. Écrire une fonction `etape_bulles : 'a array -> bool` prenant en argument un tableau et qui le parcourt et fait des échanges comme indiqué dans l'étape (a). Elle renverra `true` si au moins un échange a été effectué (et `false` sinon).
3. Écrire une fonction `tri_bulles : 'a array -> unit` prenant en argument un tableau et qui le trie par ordre croissant en utilisant le principe du tri à bulles.

Exercice 5 (tri par sélection) :
Dans cet exercice, si `t` est un tableau, et si `i` et `j` sont deux entiers, on note `t[i:j]` le sous-tableau contenant les valeurs : `t.(i)`, `t.(i+1)`, ..., `t.(j-1)`.

Le **tri par sélection** est un algorithme qui permet de trier un tableau d'éléments dans l'ordre croissant. Il repose sur le principe suivant (on note `t` le tableau et n sa longueur) :

- pour $i \in [\![0, n-2]\!]$:
  (a) on cherche l'indice du minimum de `t[i:n]` (notons $i_{\min}$ l'indice obtenu) ;
  (b) on échange `t.(`i`)` et `t.(`$i_{\min}$`)`.

1. Écrire une fonction `indice_min : 'a array -> int -> int` prenant en arguments un tableau `t` et un indice i, et renvoyant l'indice $i_{\min}$ décrit à l'étape (a).
2. Écrire une fonction `tri_selection : 'a array -> unit` prenant en argument un tableau et qui le trie par ordre croissant en utilisant le principe du tri par sélection.

Exercice 6 (tri par insertion) : Dans cet exercice, si `t` est un tableau, et si `i` et `j` sont deux entiers, on note `t[i:j]` le sous-tableau contenant les valeurs : `t.(i)`, `t.(i+1)`, ..., `t.(j-1)`.

Le **tri par insertion** est un algorithme qui permet de trier un tableau d'éléments dans l'ordre croissant. Il repose sur le principe suivant (on note `t` le tableau et n sa longueur) :

- pour $i \in [\![1, n-1]\!]$:
  - on suppose que `t[0:i]` est déjà trié, et on insère `t.(i)` au bon endroit pour que `t[0:i+1]` devienne trié (il faut trouver l'indice j où l'insérer, et décaler tous les éléments à partir de ce j d'un cran vers la droite).

1. Écrire une fonction `tri_insertion : 'a array -> unit` prenant en argument un tableau et qui le trie par ordre croissant en utilisant le principe du tri par insertion.

Exercice 7 (rendu de monnaie) :
Dans cet exercice, on souhaite implémenter un algorithme permettant de rendre la monnaie dans un système monétaire donné, en utilisant le moins de pièces possibles. On fera l'hypothèse que l'ensemble des pièces du système monétaire comporte au moins une pièce de valeur 1. Ainsi quel que soit la valeur de la monnaie qu'on veuille rendre, il existe au moins une façon de la rendre.

On modélise un système monétaire par un tableau d'entiers contenant les valeurs des pièces possibles. On supposera qu'un tel tableau est **déjà trié** par valeurs croissantes de pièces.

Par exemple, pour modéliser le système monétaire européen classique, on prendra comme unité le centime et on considèrera les pièces `p = [|1;2;5;10;20;50|]`.

Approche gloutonne. L'approche gloutonne repose sur l'idée suivante :

- pour rendre la somme, on rend la pièce de la plus grande valeur tant que c'est possible ;
- puis on passe à la deuxième pièce, etc. jusqu'à avoir rendu toute la monnaie.

1. Proposer une implémentation impérative ou récursive de cette approche.

Exemple
```
# monnaie_glouton [|1;2;5;10;20;50|] 139 ;;
- : int list = [2; 2; 5; 10; 20; 50; 50]
```

Programmation dynamique. L'approche gloutonne est optimale pour la plupart des systèmes monétaires du monde. Mais elle ne permet pas en toute généralité de trouver le nombre minimal de pièces nécessaires pour rendre la monnaie. Par exemple, l'approche gloutonne pour rendre 48 avec les pièces de l'ancien système britannique (`p = [|1;3;6;12;24;30|]`) conduit à rendre la monnaie avec 3 pièces, alors qu'il existe manifestement une solution optimale avec deux pièces uniquement.

```
# monnaie_glouton [|1;3;6;12;24;30|] 48 ;;
- : int list = [6; 12; 30]
```

On pose $s(i,k)$ le nombre minimal de pièces nécessaire pour obtenir la somme k avec les i premières pièces du système monétaire, on va remplir une matrice petit à petit avec toutes ces valeurs.

2. On suppose que les pièces sont représentées par un tableau de valeurs p. Justifier que l'équation de programmation dynamique satisfaite par s est :

$$s(i,k) = \min \begin{cases} 1 + s(i, k - p.(i)) \text{ si } (\mathrm{k} - \mathrm{p.(i)}) \geq 0 \\ s(i-1,k) \text{ si i} \geq 1 \end{cases}$$

3. Proposer une implémentation de cette méthode. Votre fonction prendra en entrée un tableau correspondant aux valeurs des pièces et un entier correspondant au montant à rendre et renverra le nombre minimal de pièces nécessaire pour rendre la monnaie.
4. Améliorer votre programme pour renvoyer la liste des pièces impliquées.

Exemple
```
# monnaie_dyna [|1;3;6;12;24;30|] 48 ;;
- : int list = [24; 24]
```